TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM

TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG

KHOA CÔNG NGHỆ THÔNG TIN

PHẠM DUY KHÁNH - 522H0064

PHẠM VĂN PHÚC - 522H0063

NGUYỄN HUỲNH ANH KHOA - 522H0046

TỐNG NGUYÊN GIA HUY -522H0077

BÀI TIỂU LUẬN

BÁO CÁO GIỮA KỲ

MẪU THIẾT KẾ

THÀNH PHỐ HỒ CHÍ MINH, NĂM 2025

TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM

TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG

KHOA CÔNG NGHỆ THÔNG TIN

PHẠM DUY KHÁNH - 522H0064

PHẠM VĂN PHÚC - 522H0063

NGUYỄN HUỲNH ANH KHOA - 522H0046

TỐNG NGUYÊN GIA HUY -522H0077

BÀI TIỂU LUẬN

BÁO CÁO GIỮA KỲ

MẪU THIẾT KẾ

Người hướng dẫn

Thầy Hà Lê Hoài Trung

THÀNH PHỐ HỒ CHÍ MINH, NĂM 2025

LỜI CẢM ƠN

Chúng em xin chân thành cảm ơn thầy Hà Lê Hoài Chung đã giảng dạy môn học Mẫu thiết kế, giúp chúng em không chỉ nắm vững kiến thức lý thuyết mà còn áp dụng vào thực tế một cách hiệu quả.

Chúng em cũng xin gửi lời cảm ơn đến thầy vì đã trực tiếp hướng dẫn và hỗ trợ chúng em trong việc hoàn thành bài báo cáo giữa kỳ. Sự hướng dẫn chi tiết của thầy đã giúp chúng em hoàn thiện bài làm của mình một cách tốt nhất.

Chúng em thật sự cảm ơn thầy và mong tiếp tục được thầy giảng dạy và hỗ trợ trong thời gian sắp tới.

TP. Hồ Chí Minh, ngày 2 tháng 4 năm 2025

Tác giả

Phạm Duy Khánh

Phạm Văn Phúc

Nguyễn Huỳnh Anh Khoa

Tống Nguyễn Gia Huy

CÔNG TRÌNH ĐƯỢC HOÀN THÀNH
TẠI TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG

Tôi xin cam đoan đây là công trình nghiên cứu của riêng tôi và được sự hướng dẫn khoa học của Thầy Hà Lê Hoài Trung. Các nội dung nghiên cứu, kết quả trong đề tài này là trung thực và chưa công bố dưới bất kỳ hình thức nào trước đây. Những số liệu trong các bảng biểu phục vụ cho việc phân tích, nhận xét, đánh giá được chính tác giả thu thập từ các nguồn khác nhau có ghi rõ trong phần tài liệu tham khảo.

Ngoài ra, trong Dự án còn sử dụng một số nhận xét, đánh giá cũng như số liệu của các tác giả khác, cơ quan tổ chức khác đều có trích dẫn và chú thích nguồn gốc.

Nếu phát hiện có bất kỳ sự gian lận nào tôi xin hoàn toàn chịu trách nhiệm về nội dung Dự án của mình. Trường Đại học Tôn Đức Thắng không liên quan đến những vi phạm tác quyền, bản quyền do tôi gây ra trong quá trình thực hiện (nếu có).

TP. Hồ Chí Minh, ngày 2 tháng 4 năm 2025

Tác giả

Phạm Duy Khánh

Phạm Văn Phúc

Nguyễn Huỳnh Anh Khoa

Tống Nguyễn Gia Huy

TIỂU LUẬN GIỮA KỲ

TÓM TẮT

Phân tích và làm rõ các mẫu thiết kế của trò chơi mô phỏng hoạt động của xe đạp và drone, tập trung vào việc điều khiển phương tiện, quản lý trạng thái và tương tác với môi trường. Các chức năng chính bao gồm bật/tắt turbo, nhận sát thương, điều khiển hướng di chuyển, gọi và điều khiển drone. Trò chơi còn có hệ thống ghi và phát lại hành động, giúp tái hiện thao tác của người chơi.

MỤC LỤC

DANH MỤC HÌNH VẼ 6
CHƯƠNG 1. GIỚI THIỆU BÀI TOÁN 1
1.1 Mô tả bài toán 1
1.2 Các chức năng chính 1
CHƯƠNG 2. LƯỢC ĐỒ CLASS THIẾT KẾ 11
2.1 Observer pattern 11
2.2 Singleton pattern 11
2.3 Strategy pattern 12
2.4 Command pattern 13
2.5 State pattern 13
CHƯƠNG 3. HIỆN THỰC MẪU 15
3.1 Observer pattern 15
3.1.1 Mục đích 15
3.1.2 Tương tác 15
3.2 State pattern 15
3.2.1 Mục đích 15
3.2.2 Tương tác 15
3.3 Strategy pattern 15
3.3.1 Mục đích 16
3.3.2 Tương tác 16
3.4 Command pattern 16
3.4.1 Mục đích 16
3.4.2 Tương tác 16
3.5 Singleton pattern 16
3.5.1 Mục đích 16
3.5.2 Tương tác 16
CHƯƠNG 4. CHỈNH SỬA CHỨC NĂNG MẪU THIẾT KẾ 17
4.1 Chức năng phục hồi cho xe 17
4.1.1 Mô tả tính năng 17
4.1.2 Áp dụng vào pattern 17

4.1.3 Cách triển khai 17
4.1.4 Tương tác giữa các thành phần 17
4.2 Chức năng khiên bảo vệ 17
4.2.1 Mô tả tính năng 17
4.2.2 Áp dụng vào pattern 18
4.2.3 Cách triển khai 18
4.2.4 Tương tác giữa các thành phần 19
4.3 Sơ đồ class các Design Pattern sau khi thêm các chức năng 20
4.3.1 Observer pattern 20
4.3.2 Strategy Pattern 21
4.3.3 Command Pattern 23
4.3.4 Singleton Pattern 23
4.3.5 State Pattern 24
TÀI LIỆU THAM KHẢO 25

DANH MỤC HÌNH VẼ

Hình 1: Sơ đồ class Observer pattern 11
Hình 2: Sơ đồ class Singleton pattern 11
Hình 3: Sơ đồ class Strategy pattern 12
Hình 4: Sơ đồ class Command pattern 13
Hình 5: Sơ đồ class State pattern 14
Hình 6: Sơ đồ class Observer pattern sau khi cập nhật 20
Hình 7: Sơ đồ class Strategy pattern sau khi cập nhật (Khiên bảo vệ) 21
Hình 8: Sơ đồ class Strategy pattern sau khi cập nhật (Chiến lược di chuyển của Drone) . 22
Hình 9: Sơ đồ class Command pattern sau khi cập nhật 23
Hình 10: Sơ đồ class Singleton pattern sau khi cập nhật 23
Hình 11: Sơ đồ State pattern sau khi cập nhật 24

CHƯƠNG 1. GIỚI THIỆU BÀI TOÁN

1.1 Mô tả bài toán

Đây là một trò chơi mô phỏng vận hành của các phương tiện (chẳng hạn như xe đạp và drone), trong đó sử dụng nhiều mẫu thiết kế phần mềm (design patterns) để quản lý hành vi và trạng thái của các đối tượng. Các thành phần chính của dự án bao gồm:

- BikeController: Điều khiển hành vi và trạng thái của xe đạp, bao gồm việc bật/tắt turbo, nhận sát thương và cập nhật các quan sát viên (Observer) như HUD hoặc Camera khi trạng thái thay đổi.
- HUD (Heads-Up Display) và Camera: Là các quan sát viên (Observers) theo dõi trạng thái của xe đạp và cập nhật giao diện người dùng, chẳng hạn như thông báo máu và trạng thái turbo, hoặc tạo hiệu ứng rung cho camera khi turbo được bật.
- Drone: Một đối tượng có thể di chuyển theo các chiến lược khác nhau (Weaving, Bopping, Fallback) mà được áp dụng thông qua mô hình Strategy Pattern.
- Command Pattern: Quản lý các lệnh (command) từ người dùng (như di chuyển xe đạp, thay đổi hướng, bật turbo) và có thể ghi lại và phát lại các lệnh này, cho phép người chơi ghi lại hành động và phát lại sau đó.
- State Pattern: Quản lý trạng thái của xe đạp, chẳng hạn như trạng thái di chuyển, dừng lại hoặc rẽ, mà không cần thay đổi mã nguồn của BikeController.

1.2 Các chức năng chính

a. Khởi động xe:

- Mô tả: Khi trò chơi bắt đầu, động cơ xe sẽ tự động khởi động, đánh dấu trạng thái xe sẵn sàng hoạt động. Khi xe khởi động, tất cả Observer (HUD, Camera) sẽ được cập nhật.

- Đối tượng thực hiện: BikeController (thuộc namespace Chapter.Observer).
- Cách thức thực hiện:
    - Chức năng này được thực hiện trong phương thức StartEngine() của lớp BikeController.
    - Cụ thể:
        - Biến _isEngineOn (private bool) được đặt thành true, thể hiện động cơ đã được khởi động.
        - Sau đó, phương thức NotifyObservers() được gọi để thông báo cho tất cả các Observer (như HUDController và CameraController) về trạng thái mới của xe.
    - Phương thức StartEngine() được gọi tự động trong Start() của BikeController, đảm bảo xe khởi động ngay khi scene bắt đầu.

b. Sử dụng Turbo

- Mô tả: Người chơi có thể sử dụng chế độ turbo để tăng tốc độ của xe, đồng thời các thành phần giao diện (HUD) và camera sẽ phản ánh trạng thái này.
- Đối tượng thực hiện: BikeController.
- Cách thức thực hiện:
    - Chức năng này được thực hiện trong phương thức ToggleTurbo() của BikeController.
    - Cụ thể:
        - Kiểm tra điều kiện: nếu động cơ đang bật (_isEngineOn là true), trạng thái IsTurboOn (public bool) sẽ được đảo ngược (từ true thành false hoặc ngược lại).
        - Sau khi thay đổi trạng thái turbo, NotifyObservers() được gọi để thông báo cho các Observer (như HUDController để hiển thị thông báo turbo và CameraController để tạo hiệu ứng rung).

- Phương thức này có thể được kích hoạt thông qua nút giao diện trong ClientObserver hoặc lệnh từ InputHandler

c. Nhận sát thương

- Mô tả: Xe có thể bị hư hỏng khi nhận sát thương từ các nguồn bên ngoài. Nếu lượng máu (health) giảm xuống dưới 0, xe sẽ bị phá hủy.
- Đối tượng thực hiện: BikeController.
- Cách thức thực hiện
    - Chức năng này được thực hiện trong phương thức TakeDamage(float amount) của BikeController.
    - Cụ thể:
        - Giá trị health (private float, khởi tạo mặc định là 100.0f) giảm đi một lượng bằng tham số amount.
        - Trạng thái turbo (IsTurboOn) tự động tắt (false) khi xe nhận sát thương.
        - NotifyObservers() được gọi để cập nhật trạng thái mới cho HUD (hiển thị máu) và Camera (tắt hiệu ứng rung nếu có).
        - Nếu health nhỏ hơn 0, xe bị phá hủy bằng cách gọi Destroy(gameObject), xóa đối tượng xe khỏi scene.
    - Phương thức này có thể được gọi từ giao diện trong ClientObserver khi nhấn nút "Damage Bike".

d. Hiển thị thông tin trên HUD

- Mô tả: HUD (Heads-Up Display) hiển thị thông tin về mức máu của xe, trạng thái turbo, và cảnh báo khi máu xuống thấp để người chơi nắm bắt tình hình.
- Đối tượng thực hiện: HUDController (thuộc namespace Chapter.Observer)
- Cách thức thực hiện:

- Chức năng này được thực hiện trong phương thức OnGUI() của HUDController.
- Cụ thể:
    - Hiển thị mức máu hiện tại (_currentHealth) trong một hộp giao diện với nhãn "Health: ".
    - Nếu turbo đang bật (_isTurboOn là true), hiển thị thông báo "Turbo Activated!" trong một hộp giao diện khác.
    - Nếu máu giảm xuống dưới hoặc bằng 50 (_currentHealth <= 50.0f), hiển thị cảnh báo "WARNING: Low Health" trong một hộp giao diện.
    - Các giá trị _isTurboOn và _currentHealth được cập nhật thông qua phương thức Notify(Subject subject), lấy dữ liệu từ BikeController khi có thay đổi trạng thái (do NotifyObservers() từ BikeController kích hoạt).

e. Hiệu ứng rung chuyển ở camera

- Mô tả: Khi turbo được bật, camera sẽ rung để tạo cảm giác tốc độ. Khi turbo tắt, camera trở về vị trí ban đầu.
- Đối tượng thực hiện: CameraController (thuộc namespace Chapter.Observer).
- Cách thức thực hiện:
    - Chức năng này được thực hiện trong phương thức Update() của CameraController.
    - Cụ thể:
        - Nếu _isTurboOn là true, vị trí của camera (transform.localPosition) được điều chỉnh ngẫu nhiên bằng cách cộng thêm một vector ngẫu nhiên (Random.insideUnitSphere * _shakeMagnitude), với _shakeMagnitude mặc định là 0.1f, tạo hiệu ứng rung.

- Nếu _isTurboOn là false, camera trở về vị trí ban đầu (_initialPosition), được lưu trong OnEnable().
- Trạng thái _isTurboOn được cập nhật trong phương thức Notify(Subject subject), lấy từ BikeController khi có thay đổi trạng thái.

f. Điều khiển xe

- Mô tả: Người chơi có thể điều khiển xe rẽ trái hoặc rẽ phải thông qua các phím nhập liệu, sử dụng Command Pattern để quản lý lệnh.
- Đối tượng thực hiện: InputHandler (namespace Chapter.Command) và các lớp lệnh TurnLeft, TurnRight.
- Cách thức thực hiện:
  - Trong InputHandler:
    - Các lệnh TurnLeft và TurnRight được khởi tạo trong Start(), liên kết với BikeController.
    - Khi người chơi nhấn phím A (rẽ trái) hoặc D (rẽ phải) và thả phím (Input.GetKeyUp), ExecuteCommand của Invoker được gọi với lệnh tương ứng, nhưng chỉ khi đang ghi hình (_isRecording là true và _isReplaying là false).
  - Trong TurnLeft và TurnRight:
    - Kế thừa từ lớp trừu tượng Command, mỗi lớp thực hiện phương thức Execute() bằng cách gọi Turn(Direction direction) của BikeController với tham số Direction.Left (-1) hoặc Direction.Right (1).
  - Mã nguồn hiện tại chưa định nghĩa phương thức Turn(Direction direction) trong BikeController, cần thêm để hoàn thiện chức năng này.

g. Ghi hình hành động của người chơi

- Mô tả: Hệ thống ghi lại các hành động của người chơi (rẽ trái, rẽ phải, bật turbo) và có thể phát lại sau đó để tái hiện.

- Đối tượng thực hiện: Invoker và InputHandler (namespace Chapter.Command).
- Cách thức thực hiện:
    - Ghi hình:
        - Khi nhấn nút "Start Recording" trong OnGUI() của InputHandler, _isRecording được đặt thành true, _isReplaying thành false, và Invoker.Record() được gọi để đặt lại _recordingTime về 0.
        - Trong Update() của InputHandler, khi nhấn phím A, D, hoặc W, các lệnh tương ứng (_buttonA, _buttonD, _buttonW) được gửi đến Invoker.ExecuteCommand().
        - Trong Invoker.ExecuteCommand(), nếu _isRecording là true, lệnh được thêm vào _recordedCommands (SortedList) với thời gian hiện tại (_recordingTime).
        - _recordingTime tăng dần trong FixedUpdate() khi đang ghi hình.
    - Dừng ghi hình: Nhấn "Stop Recording" trong OnGUI() đặt _isRecording thành false.
    - Phát lại:
        - Khi nhấn "Start Replay", _isReplaying thành true, _replayTime đặt lại về 0, và Invoker.Replay() được gọi.
        - Trong FixedUpdate() của Invoker, nếu _isReplaying là true, _replayTime tăng dần. Khi _replayTime khớp với thời gian của lệnh đầu tiên trong _recordedCommands, lệnh đó được thực thi và xóa khỏi danh sách.

h. Chuyển đổi trạng thái xe

- Mô tả: Xe có thể chuyển đổi giữa các trạng thái khác nhau (ví dụ: idle, moving, turbo, damaged) để điều khiển hành vi phù hợp.
- Đối tượng thực hiện: BikeStateContext (namespace Chapter.State).

- Cách thức thực hiện:
    - BikeStateContext giữ trạng thái hiện tại của xe trong thuộc tính CurrentState (interface IBikeState).
    - Có hai phương thức chuyển đổi:
        - Transition(): Gọi Handle() của trạng thái hiện tại (CurrentState) để thực hiện hành vi tương ứng với BikeController.
        - Transition(IBikeState state): Thay đổi CurrentState thành trạng thái mới và gọi Handle() của trạng thái đó.
    - Lưu ý: Mã nguồn hiện tại chưa triển khai các trạng thái cụ thể (như IdleState, TurboState), nhưng cấu trúc này cho phép mở rộng.

i. Gọi drone

- Mô tả: Người chơi có thể tạo ra một drone trong scene bằng cách nhấn nút "Spawn Drone", drone sẽ xuất hiện ở vị trí ngẫu nhiên.
- Đối tượng thực hiện: ClientStrategy (namespace Chapter.Strategy).
- Cách thức thực hiện:
    - Trong OnGUI() của ClientStrategy, khi nhấn nút "Spawn Drone", phương thức SpawnDrone() được gọi.
    - Trong SpawnDrone():
        - Một GameObject mới được tạo bằng GameObject.CreatePrimitive(PrimitiveType.Cube) để đại diện cho drone.
        - Component Drone được thêm vào drone bằng AddComponent<Drone>().
        - Vị trí của drone được đặt ngẫu nhiên trong một hình cầu bán kính 10 (Random.insideUnitSphere * 10).
        - Gọi ApplyRandomStrategies() để áp dụng một chiến lược di chuyển ngẫu nhiên cho drone.

j. Điều khiển drone

- Mô tả: Drone có thể di chuyển theo các chiến lược khác nhau như bay lên xuống (bopping), bay zigzag (weaving), hoặc lùi lại (fallback).
- Đối tượng thực hiện: Drone và các lớp chiến lược (BoppingManeuver, WeavingManeuver, FallbackManeuver) ,(namespace Chapter.Strategy).
- Cách thức thực hiện:
    - Trong Drone:
        - Phương thức ApplyStrategy(IManeuverBehaviour strategy) gọi Maneuver(this) của chiến lược được truyền vào để điều khiển di chuyển.
    - Các chiến lược:
        - BoppingManeuver:
            - Drone di chuyển lên xuống giữa vị trí ban đầu và độ cao tối đa (maxHeight).
            - Sử dụng StartCoroutine(Bopple(drone)) để thực hiện chuyển động mượt mà bằng Vector3.Lerp.
        - WeavingManeuver:
            - Drone di chuyển zigzag sang trái và phải với khoảng cách weavingDistance.
            - Sử dụng StartCoroutine(Weave(drone)) để thực hiện.
        - FallbackManeuver:
            - Drone lùi lại một khoảng cách fallbackDistance trên trục Z.
            - Sử dụng StartCoroutine(Fallback(drone)) để thực hiện.
    - Các chiến lược được chọn ngẫu nhiên trong ApplyRandomStrategies() của ClientStrategy.

k. Phát hiện vật cản

- Mô tả: Drone sử dụng raycast để phát hiện vật cản phía trước và có thể thay đổi hành vi di chuyển nếu cần.
- Đối tượng thực hiện: Drone (namespace Chapter.Strategy).
- Cách thức thực hiện:
    - Trong Update() của Drone:
        - Một raycast được bắn từ vị trí của drone theo hướng _rayDirection (được khởi tạo trong Start() với góc -45 độ và khoảng cách rayDistance là 15.0f).
        - Nếu raycast chạm vào một collider (Physics.Raycast), đường ray được vẽ màu xanh lục để kiểm tra; nếu không, màu xanh dương.
        - Hiện tại, mã nguồn chưa thay đổi chiến lược khi phát hiện vật cản, nhưng có thể mở rộng bằng cách gọi ApplyStrategy() với FallbackManeuver.

m. **Quản lý vòng đời game**

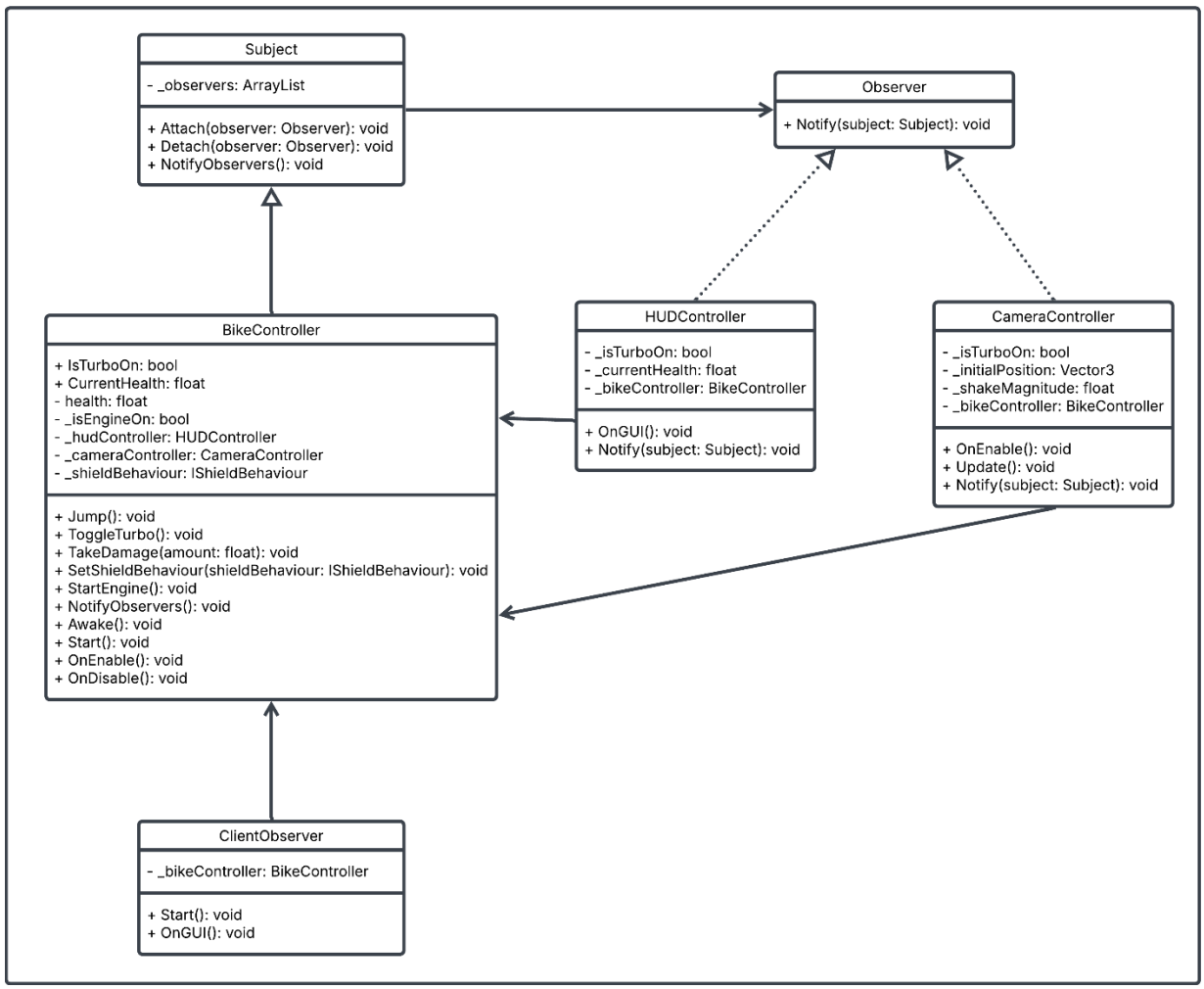
- Mô tả: GameManager theo dõi thời gian bắt đầu và kết thúc phiên chơi, hiển thị tổng thời gian chơi khi thoát game.
- Đối tượng thực hiện: GameManager (namespace Chapter.Singleton).
- Cách thức thực hiện:
    - Trong Start():
        - _sessionStartTime được đặt thành thời gian hiện tại (DateTime.Now), đánh dấu bắt đầu phiên chơi.
    - Trong OnApplicationQuit():
        - _sessionEndTime được đặt thành thời gian hiện tại.
        - Tính toán thời gian chơi bằng cách trừ _sessionStartTime từ _sessionEndTime và hiển thị qua Debug.Log.

n. Chuyển cảnh

- Mô tả: Người chơi có thể chuyển sang cảnh tiếp theo bằng cách nhấn nút "Next Scene".
- Đối tượng thực hiện: GameManager.
- Cách thức thực hiện:
    - Trong OnGUI() của GameManager, khi nhấn nút "Next Scene":
        - SceneManager.LoadScene() được gọi với chỉ số build index của cảnh hiện tại cộng thêm 1, tải cảnh tiếp theo trong danh sách build settings.

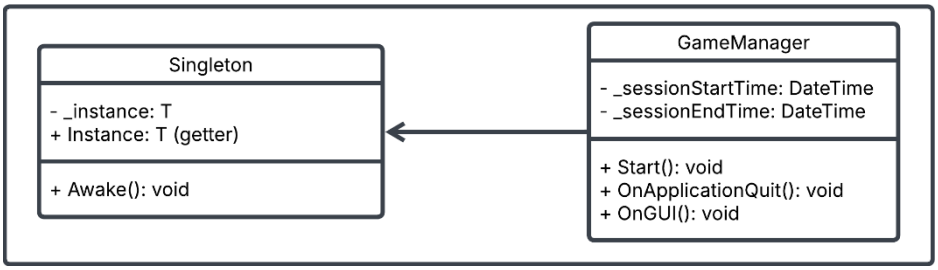
CHƯƠNG 2. LƯỢC ĐỒ CLASS THIẾT KẾ

2.1 Observer pattern



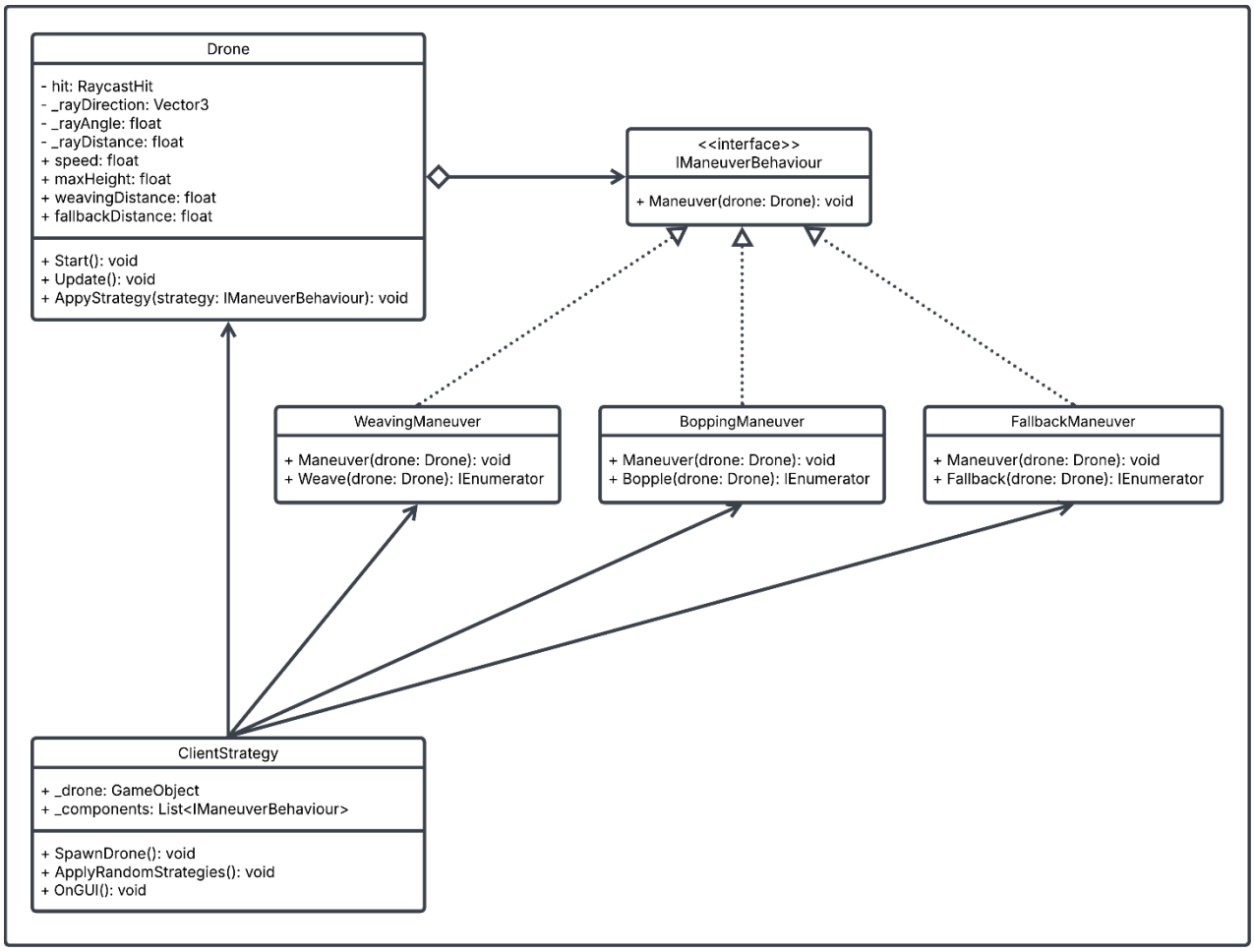
Hình 1: Sơ đồ class Observer pattern

2.2 Singleton pattern



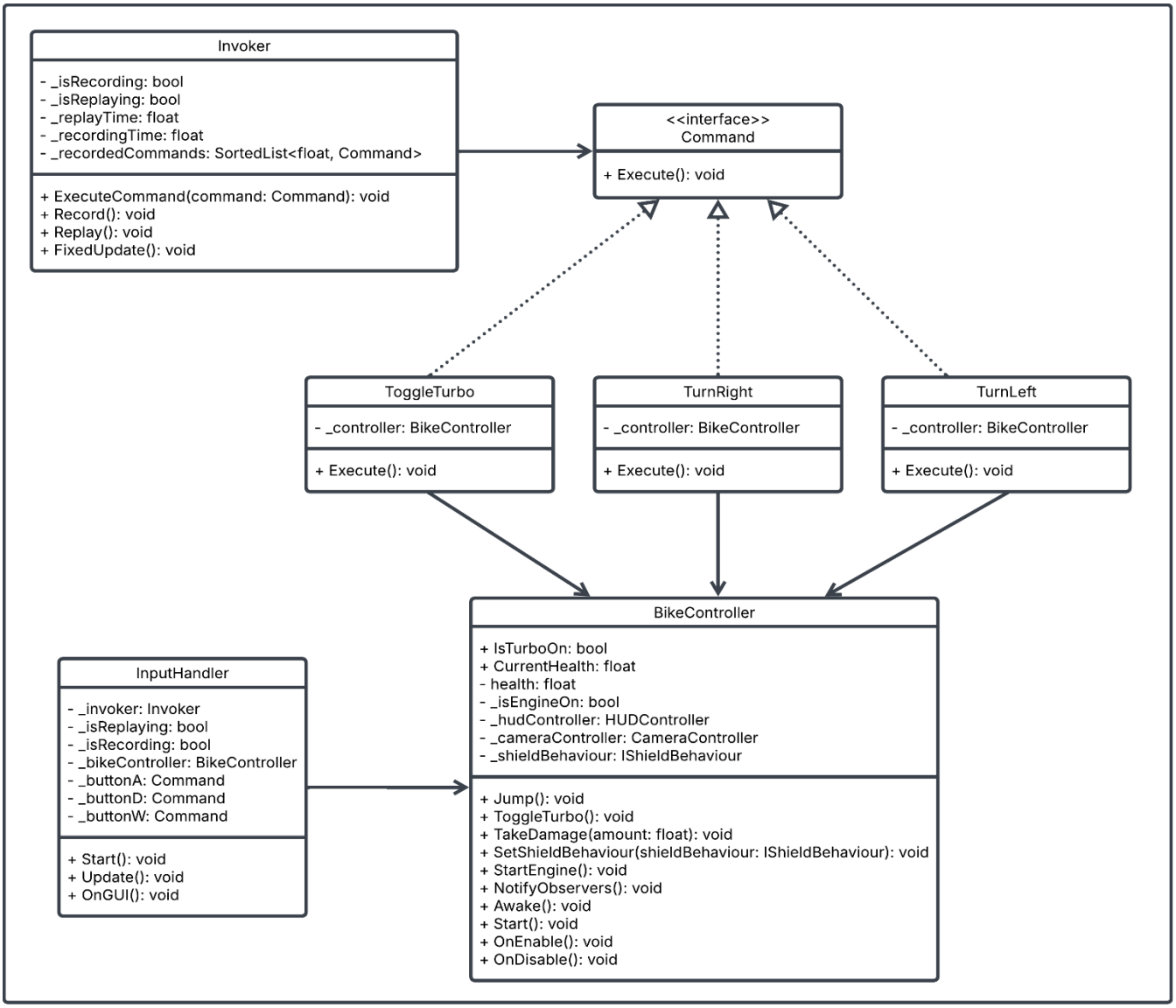
Hình 2: Sơ đồ class Singleton pattern

2.3 Strategy pattern



Hình 3: Sơ đồ class Strategy pattern

2.4 Command pattern



Hình 4: Sơ đồ class Command pattern

2.5 State pattern

Có thể nhận thấy các thành phần của **State Pattern** đã được phát hiện và triển khai một phần, tuy nhiên chưa hoàn thiện. Các thành phần hiện tại bao gồm:

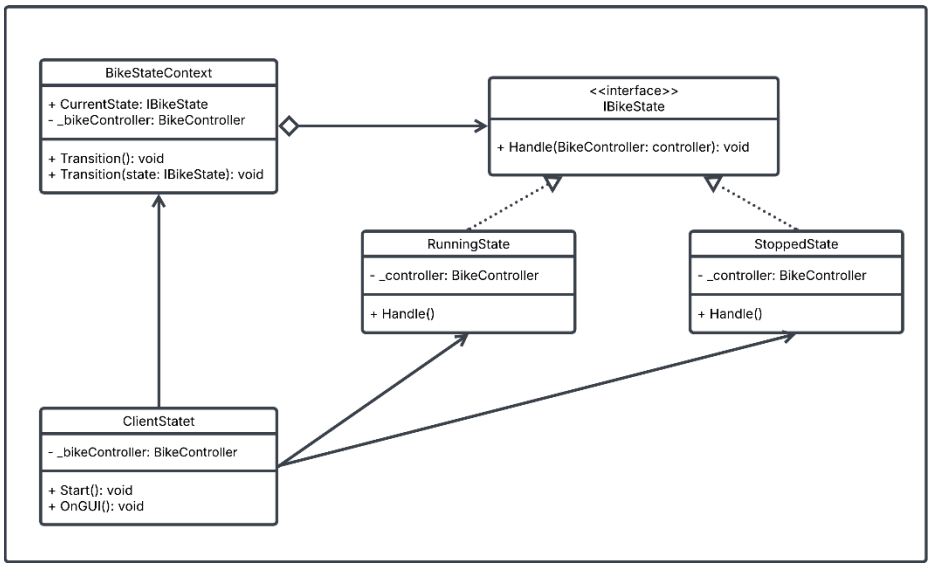
- **BikeStateContext**: Đây là lớp quản lý trạng thái của đối tượng BikeController và có khả năng chuyển đổi giữa các trạng thái khác nhau thông qua phương thức Transition(). Tuy nhiên, ở thời điểm hiện tại, nó chưa triển khai bất kỳ

trạng thái nào thực tế mà chỉ có thể đặt trạng thái mới mà không thực hiện bất kỳ thay đổi hay hành động nào.

- **IBikeState**: Đây là giao diện định nghĩa phương thức Handle() mà các trạng thái cụ thể phải thực hiện. Mỗi trạng thái sẽ có cách xử lý khác nhau cho BikeController.

Giả sử chúng muốn hoàn thiện **State Pattern** cho BikeController, với hai trạng thái cụ thể là Running và Stopped, sơ đồ lớp có thể như sau:

1. **BikeStateContext**: Lớp này sẽ giữ tham chiếu đến trạng thái hiện tại (IBikeState) và chuyển đổi giữa các trạng thái này khi cần thiết.
2. **IBikeState**: Giao diện này sẽ định nghĩa phương thức Handle() mà các lớp trạng thái phải triển khai.
3. **RunningState**: Một trạng thái cụ thể khi xe đang chạy. Lớp này sẽ triển khai phương thức Handle() để thực hiện hành động liên quan đến trạng thái xe đang chạy.
4. **StoppedState**: Một trạng thái cụ thể khi xe dừng lại. Lớp này sẽ triển khai phương thức Handle() để thực hiện hành động liên quan đến trạng thái xe dừng lại.



Hình 5: Sơ đồ class State pattern

CHƯƠNG 3. HIỆN THỰC MẪU

3.1 Observer pattern

3.1.1 Mục đích

Cho phép các đối tượng (observers) nhận thông báo từ một đối tượng chủ thể (subject) khi trạng thái thay đổi.

3.1.2 Tương tác

- BikeController là subject, khi trạng thái thay đổi (bật turbo qua ToggleTurbo() hoặc mất máu qua TakeDamage()), nó gọi NotifyObservers() để thông báo cho các Observer đã đăng ký.
- Các observer (HUDController, CameraController) đăng ký qua Attach() và nhận thông báo qua Notify().
- HUDController cập nhật giao diện (health, turbo), còn CameraController điều chỉnh camera (rung khi turbo bật) khi nhận thông báo từ BikeController .

3.2 State pattern

3.2.1 Mục đích

Thay đổi hành vi của xe đạp dựa trên trạng thái hiện tại.

3.2.2 Tương tác

- BikeStateContext giữ tham chiếu đến CurrentState (một instance của IBikeState).
- Khi gọi phương thức Transition() hoặc Transition(IBikeState), trạng thái hiện tại của BikeStateContext được thay đổi.
- Sau khi thay đổi trạng thái, phương thức Handle() của trạng thái mới được gọi để thực hiện hành vi tương ứng (ví dụ: chạy, dừng).

3.3 Strategy pattern

3.3.1 Mục đích

Định nghĩa các thuật toán di chuyển cho drone, cho phép hoán đổi linh hoạt.

3.3.2 Tương tác

- Drone gọi ApplyStrategy(IManeuverBehaviour) để áp dụng chiến lược.
- Các chiến lược (BoppingManeuver, WeavingManeuver, FallbackManeuver) thực hiện hành vi cụ thể thông qua Maneuver() (ví dụ: nhún nhảy, lượn sóng).

3.4 Command pattern

3.4.1 Mục đích

Đóng gói các lệnh điều khiển xe đạp, hỗ trợ ghi và phát lại.

3.4.2 Tương tác

- InputHandler nhận input (A, D, W) và tạo lệnh (TurnLeft, TurnRight, ToggleTurbo).
- Invoker ghi lại lệnh với thời gian (_recordedCommands) và phát lại theo thứ tự khi gọi Replay().

3.5 Singleton pattern

3.5.1 Mục đích

Đảm bảo chỉ có một instance của GameManager tồn tại.

3.5.2 Tương tác

- Singleton<T> kiểm tra và tạo instance duy nhất qua Instance.
- GameManager sử dụng để quản lý phiên chơi (thời gian bắt đầu/kết thúc, chuyển cảnh).

CHƯƠNG 4. CHỈNH SỬA CHỨC NĂNG MẪU THIẾT KẾ

4.1 Chức năng phục hồi cho xe

4.1.1 Mô tả tính năng

Thêm khả năng hồi phục máu cho xe đạp bằng một lệnh mới. Xe có thể hồi phục máu thông qua một lệnh HealCommand được gửi tới BikeController.

4.1.2 Áp dụng vào pattern

Command Pattern sẽ bị thay đổi:

- Command Pattern rất phù hợp vì nó giúp ghi lại hành động hồi phục máu và có thể phát lại nếu cần.
- Dễ dàng tích hợp vào hệ thống Command hiện tại(Invoker, BikeController).

4.1.3 Cách triển khai

- Tạo Command mới: HealCommand
- Thêm phương thức Heal trong BikeController
- Thêm vào InputHandler

4.1.4 Tương tác giữa các thành phần

- InputHandler nhận lệnh hồi phục máu
- Invoker thực thi lệnh HealCommand, gọi phương thức Heal() trong BikeController.
- BikeController cập nhật giá trị health và gọi NotifyObservers() để thông báo.
- HUDController nhận thông báo và cập nhật giá trị máu trên giao diện.

4.2 Chức năng khiên bảo vệ

4.2.1 Mô tả tính năng

Các vật phẩm khiên bảo vệ sẽ xuất hiện ngẫu nhiên trên đường đua. Khi xe va chạm với vật phẩm khiên, xe sẽ nhận khiên bảo vệ, Trang bị khiên bảo vệ sẽ giúp giảm sát thương nhận vào từ các nguồn khác nhau.

Có thể thu thập khiên từ những vật phẩm ngẫu nhiên. Có 3 loại khiên và hiệu ứng lần lượt của từng loại khiên là:

- BasicShield : Có tác dụng làm giảm 20% sát thương nhận vào.
- PlatinumShield: Có tác dụng làm giảm 50% sát thương nhận vào.
- AbsoluteShield: Có tác dụng miễn 100% sát thương và hiệu ứng bất lợi phải nhận.

4.2.2 Áp dụng vào pattern

Sử dụng Strategy Pattern. Vì

- Dễ dàng thay đổi hoặc thêm các loại khiên bảo vệ khác nhau như BasicShield, PlatinumShield, AbsoluteShield.
- Thay đổi hành vi nhận sát thương mà không cần thay đổi lớp BikeController, vì việc xử lý giảm sát thương đã được phân tách thành các chiến lược khiên cụ thể.
- Việc thay đổi các chiến lược khiên (ví dụ, khi thay đổi loại khiên) không ảnh hưởng đến các lớp khác, chỉ cần thay đổi chiến lược mà BikeController sử dụng.

Ngoài ra pattern cũng có sự thay đổi để hiện thị loại khiên đang được áp dụng lên xe.

4.2.3 Cách triển khai

Tạo Interface IShieldBehaviour:

- Triển khai các loại khiên khác nhau như BasicShield, PlatinumShield, AbsoluteShield

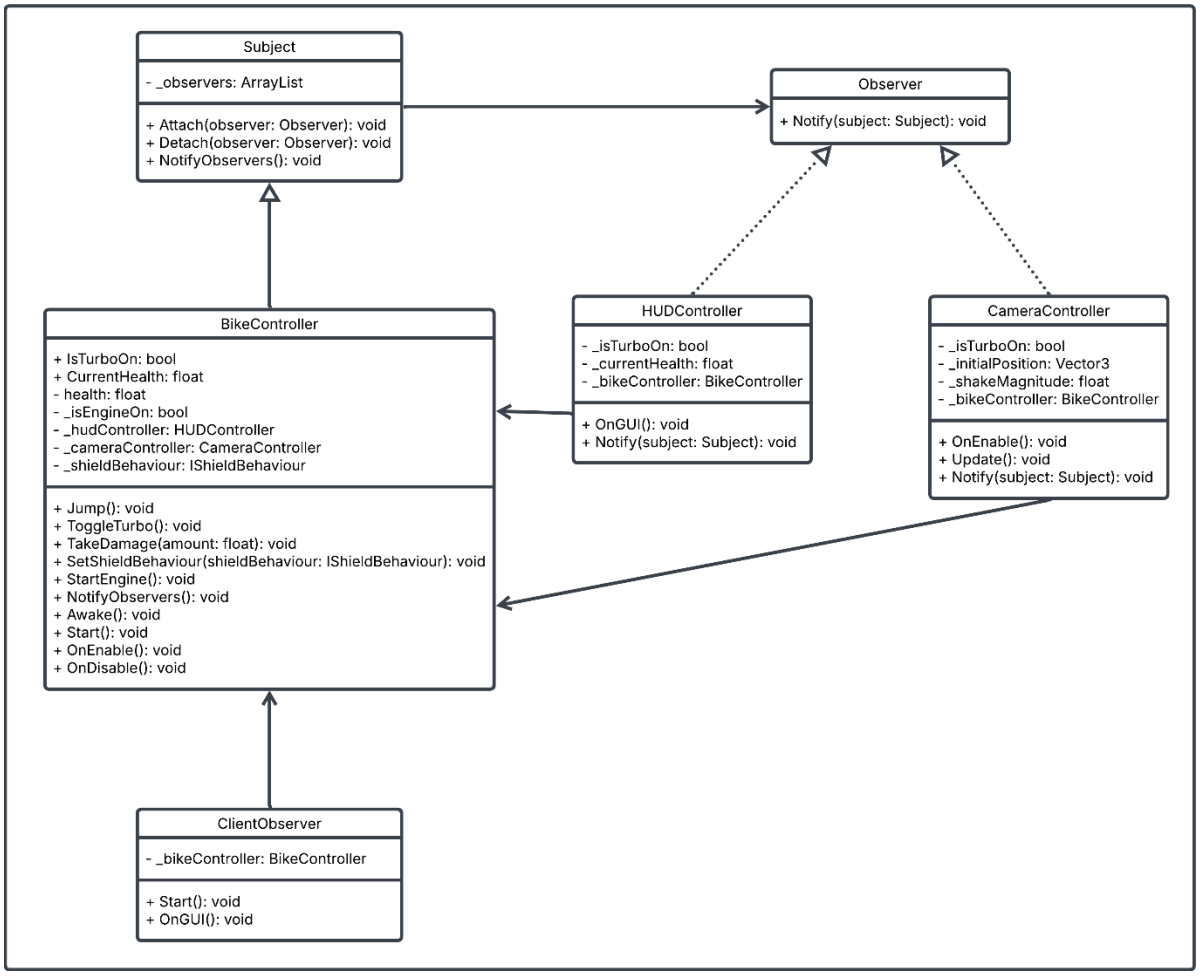
- Tạo class ShieldItem: là lớp đại diện cho mỗi vật phẩm khiên xuất hiện trên đường đua. Khi BikeController va chạm với vật phẩm này, chiến lược khiên được gán cho BikeController.
- Tạo class ShieldItemManager là lớp tạo ra vật phẩm khiên ngẫu nhiên trên đường đua. Mỗi vật phẩm khiên sẽ có một loại khiên ngẫu nhiên (Basic, Platinum, Absolute) và sẽ gán chiến lược khiên cho BikeController khi va chạm. Lớp này không trực tiếp nằm trong Strategy Pattern vì nó chỉ có chức năng sinh ra các vật phẩm khiên ngẫu nhiên và không thay đổi hành vi xử lý sát thương, có tác dụng hỗ trợ cho class ShieldItem.
- Tích hợp vào BikeController: BikeController là lớp chính để điều khiển xe đạp và nhận sát thương. Nó sử dụng IShieldBehaviour để giảm sát thương nhận vào.

4.2.4 Tương tác giữa các thành phần

- ShieldItemManager là lớp chịu trách nhiệm sinh ra các vật phẩm khiên ngẫu nhiên trên đường đua. Lớp này sẽ tạo ra các vật phẩm khiên theo một khoảng thời gian định kỳ (spawnInterval) và xuất hiện trong phạm vi ngẫu nhiên trên đường đua (spawnRange).
- ShieldItem là lớp đại diện cho mỗi vật phẩm khiên khi sinh ra trên đường đua. Lớp này có thể là một đối tượng trong game như một quả cầu hoặc hộp, khi xe đạp (BikeController) va chạm với vật phẩm này, nó sẽ nhận khiên bảo vệ.
- Khi xe đạp va chạm với vật phẩm khiên, BikeController sẽ nhận được khiên bảo vệ từ vật phẩm đó và chiến lược khiên sẽ được gán cho BikeController thông qua phương thức SetShieldBehaviour(). Khi đó, BikeController sẽ có khả năng giảm sát thương nhận vào tùy theo loại khiên (ví dụ: BasicShield, PlatinumShield, AbsoluteShield).

4.3 Sơ đồ class các Design Pattern sau khi thêm các chức năng

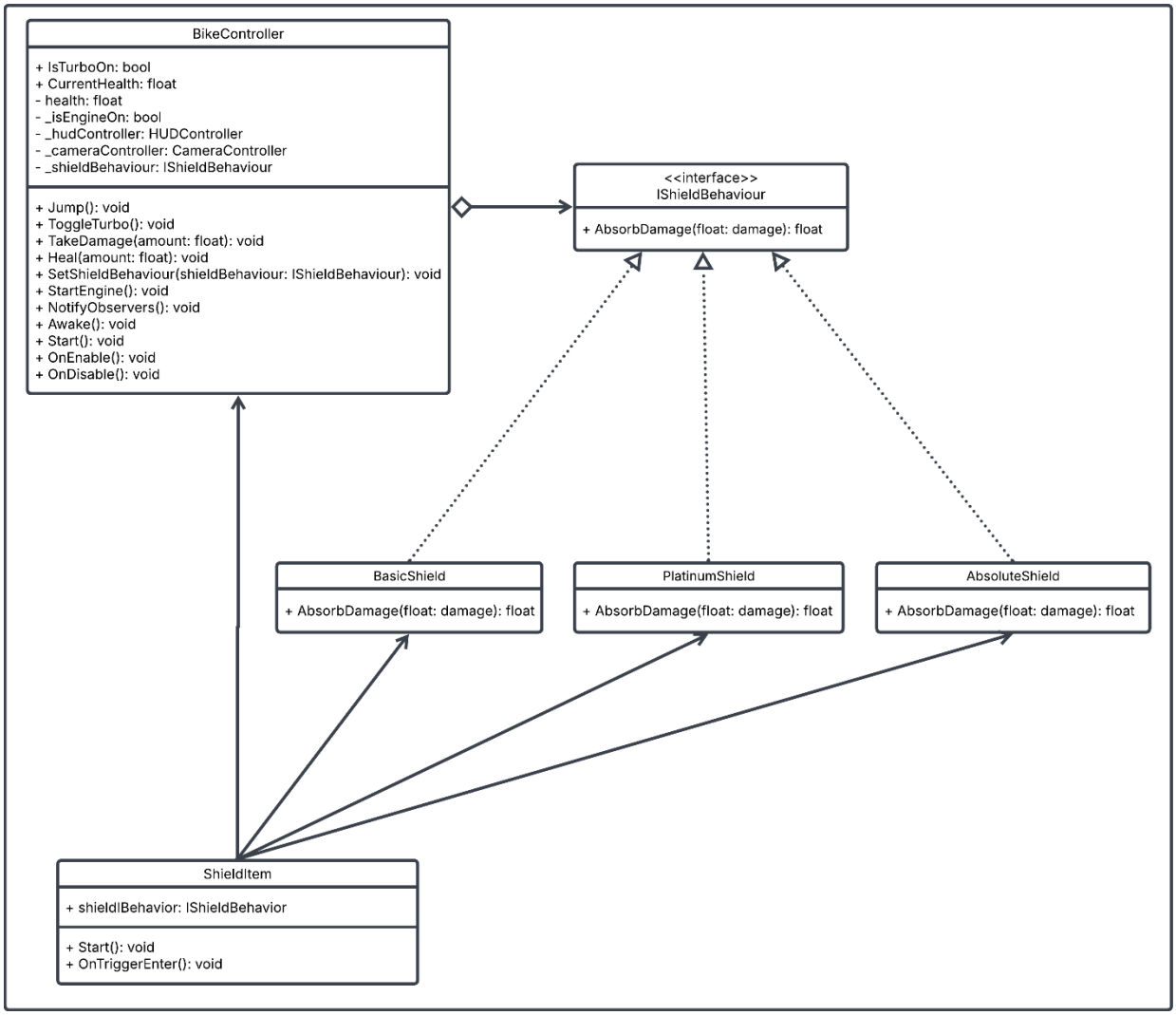
4.3.1 Observer pattern



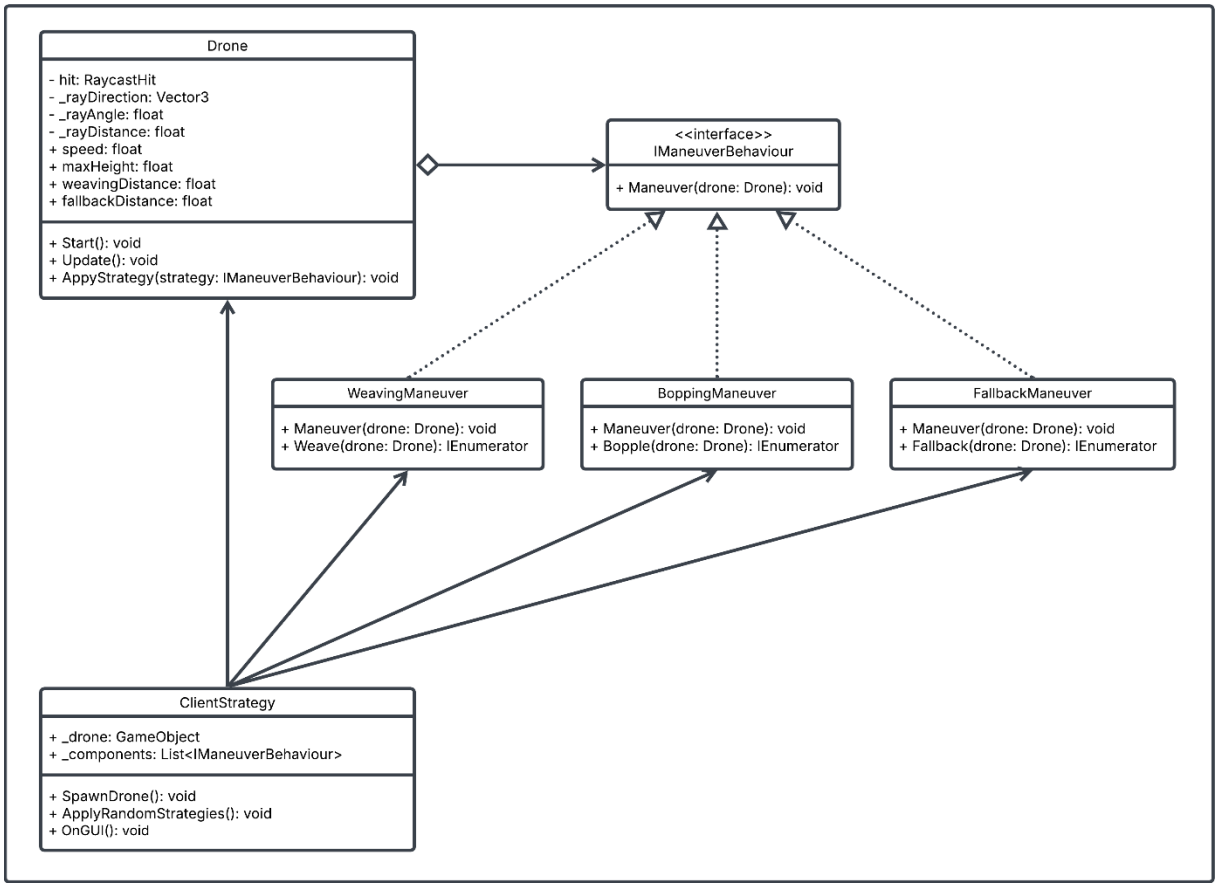
Hình 6: Sơ đồ class Observer pattern sau khi cập nhật

4.3.2 Strategy Pattern

Strategy Pattern có thêm phần khiên bảo vệ cho xe. Dù phần chiến lược di chuyển của Drone và phần khiên bảo vệ cho xe không có mối liên hệ trực tiếp với nhau, nhưng cả hai đều áp dụng Strategy Pattern để linh hoạt thay đổi hành vi hoặc chiến lược trong các tình huống khác nhau. Strategy Pattern giúp hệ thống trở nên dễ bảo trì, mở rộng và dễ dàng thay đổi các chiến lược mà không làm ảnh hưởng đến mã nguồn của các lớp khác. Vì vậy nên Strategy Pattern sẽ gồm hai phần là chiến lược di chuyển của Drone và phần khiên bảo vệ cho xe.

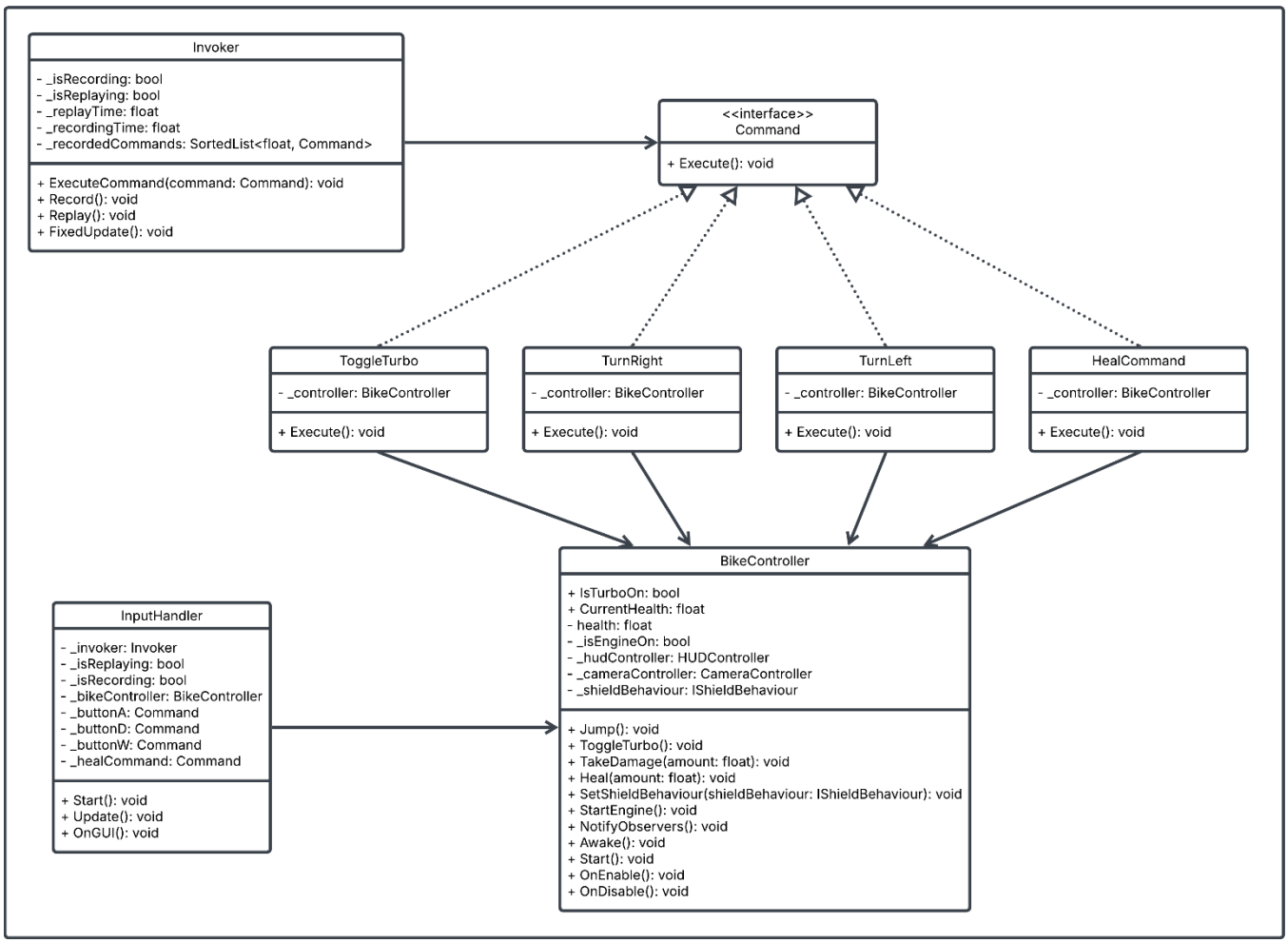


Hình 7: Sơ đồ class Strategy pattern sau khi cập nhật (Khiên bảo vệ)



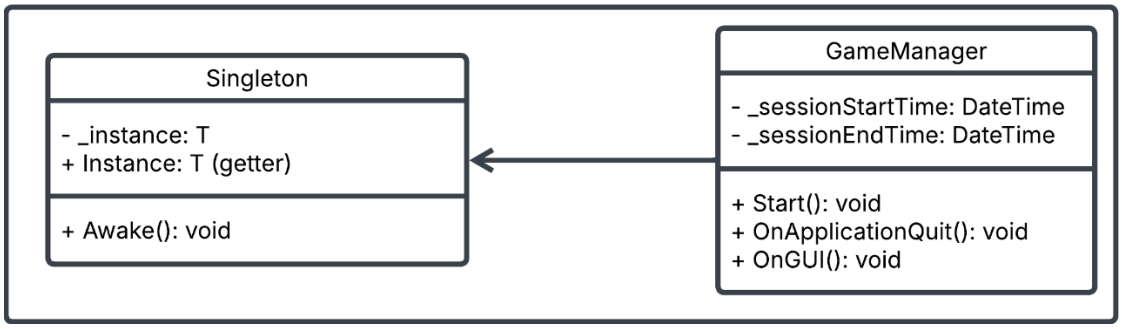
Hình 8: Sơ đồ class Strategy pattern sau khi cập nhật (Chiến lược di chuyển của Drone)

4.3.3 Command Pattern



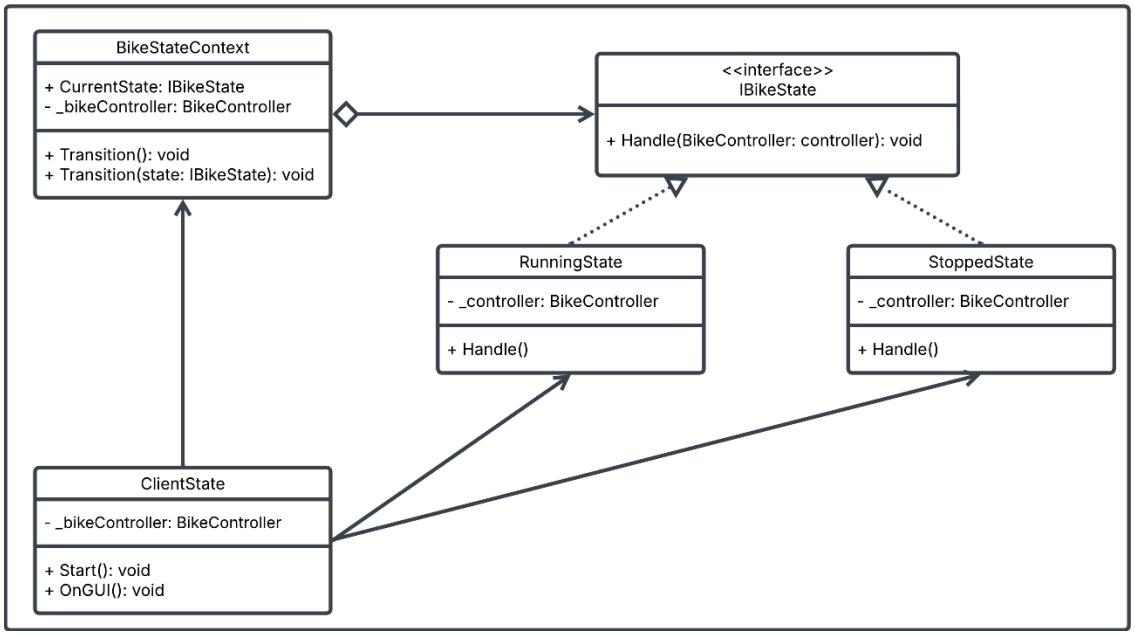
Hình 9: Sơ đồ class Command pattern sau khi cập nhật

4.3.4 Singleton Pattern



Hình 10: Sơ đồ class Singleton pattern sau khi cập nhật

4.3.5 State Pattern



Hình 11: Sơ đồ State pattern sau khi cập nhật

TÀI LIỆU THAM KHẢO

1. Strategy pattern by Geeksforgeeks
2. Singleton pattern by Geeksforgeeks
3. State pattern by Geeksforgeeks
4. Observer pattern by Geeksforgeeks
5. Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra, [2004], Head First Design Patterns, O'Reilly Media, Sebastopol.
6. Steven John Metsker, William C. Wake, [2006], Design Patterns in Java, Addison-Wesley, New Jersey.
7. Erich Gamma, John Vlissides, Ralph Johnson, Richard Helm, [1995], Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Boston.
8. Christopher G. Lasater, [2007], Design Patterns, Worldware Publications, Texas.